Riassunto modulo 2 (A.A 2019-2020)

Modelli e Processi

Modello

Per modello si intende genericamente una rappresentazione di un oggetto reale che riproduce caratteristiche o comportamenti ritenuti fondamentali o utili per il tipo di ricerca che si sta svolgendo.
Per quanto riguarda l'ingegneria del software è un insieme di proprietà e caratteristiche espresse per mezzo di un linguaggio apposito e diagrammi.
Lo scopo dei diagrammi è descrivere in maniera concisa e precisa conoscenze sul problema utili in modo da individuare rischi e scelte progettuali. Si usano linguaggi con livello di astrazione più elevato dei linguaggi di programmazione.
L'insieme dei modelli, connessi in modo sistematico, deve dare una descrizione completa e non ridondante e la transizione tra un modello e l'altro deve essere continua.

N.B. modello != rappresentazione

Tracciabilità

In qualsiasi direzione si percorra la sequenza di modelli generati, deve essere possibile mappare uno o più elementi in un modello in uno o più elementi in un altro. Essa:

- garantisce consistenza e coerenza
- permette di tenere sotto controllo le modifiche
- creare un percorso logico tra i requisiti e la soluzione

Linguaggio di modellazione

È un linguaggio semi-formale che può essere usato per modellare/descrivere un sistema. Tramite tale linguaggio si ottiene una rappresentazione del modello, differente a seconda del linguaggio usato (il potere espressivo dei vari linguaggi può variare), che che può essere utilizzata per comunicare tra i gli addetti ai lavori in maniera possibilmente precisa e flessibile.

Anche il codice permette di ottenere una rappresentazione di un modello, ma questa rappresentazione ha vari svantaggi:

- è troppo dettagliata
- fornisce una visione piatta
- non mette in evidenza i punti salienti
- non permette di avere rapidamente una visione di insieme

Dato che è necessario avere sia modelli ad alto livello che codice (funzionante) diventa necessario mantenere l'allineamento tra modello e codice (non si usano tecniche di *reverse engineering* in quanto riproducono nel modello gli stessi problemi del codice stesso). In caso l'allineamento non venga mantenuto viene meno la tracciabilità e i modelli non forniscono più una vista coerente.
L'approccio migliore consiste nell'apportare modifiche in primo luogo al modello di progettazione e generare da questo, dove possibile, del codice.

Processi di sviluppo

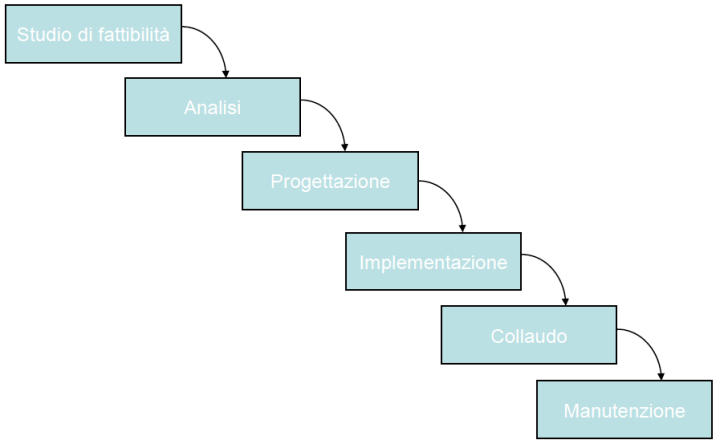
È un insieme ordinato di passi che coinvolge tutte quelle attività, vincoli e risorse necessari per produrre il desiderato output a partire dall'insieme dei requisiti in ingresso. Tipicamente è composto da diverse fasi relazionate tra loro. Ognuna di esse identifica una porzione di lavoro da svolgere con le relative risorse ed i relativi vincoli.
Le fasi generiche sono:

1. Specifica: riguarda cosa il sistema debba fare, quali siano i vincoli
2. Sviluppo: riguarda la produzione del sistema
3. Validazione: riguarda la verifica che quanto creato rispetti le specifiche del committente
4. Evoluzione: riguarda i cambiamenti nel prodotto in accordo alle modifiche richieste dal committente

Modelli di processo

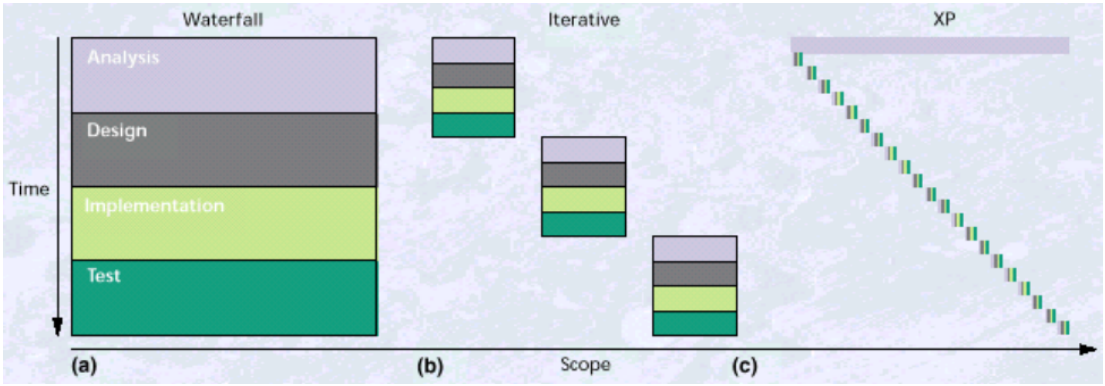
Un modello di processo definisce uno schema (template) attorno a cui organizzare un processo di sviluppo. Prescrive le fasi attorno alle quali è organizzato il processo, in quale ordine esse vadano seguite e quali siano le relazioni tra di esse.

- **Modello a cascata**: ogni fase deve essere svolta in maniera esaustiva prima di passare alla successiva, in quanto non è possibile tornare ad una fase precedente (anche se alcune varianti permettono di tornare alla fase immediatamente precedente per rivedere alcuni dettagli). Il prodotto di ogni fase è detto **semilavorato**.



  Questo modello non è adatto a progetti con requisiti instabili a causa della sua rigidità. Nella realtà infatti i requisiti dei committenti evolvono nel tempo, pertanto per evitare di incorrere in problemi è meglio realizzare un prototipo su cui lavorare inizialmente in modo da fornire una prima dimostrazione al committente e da capire i requisiti al meglio. Tale prototipo sarà poi abbandonato una volta che il sistema sarà costruito nel suo complesso. Questo approccio risulta però quasi sempre così dispendioso da annullare i vantaggi economici che il modello a cascata dovrebbe garantire.

- **Modello evolutivo**: si parte dallo sviluppo di un prototipo iniziale che viene mano a mano raffinato, partendo dalle parti con requisiti più chiari, passando per versioni intermedie in cui si aggiungono i requisiti richiesti dai committenti, fino ad arrivare al prodotto finale. Questo permette di lavorare a stretto contatto con gli utenti in modo da comprenderne al meglio le specifiche e le funzionalità. Mano a mano che le iterazioni si fanno più veloci ci avvicina alla *extreme programming* (XP) cioè si progetta mentre si sviluppa, rende necessario mantenere una certa semplicità nel codice e la scrittura di molti test. A causa delle modifiche frequenti il sistema sviluppato è poco strutturato e non è visibile il processo di sviluppo, perciò questo modello è adatto a progetti di piccole dimensioni o che hanno breve durata.



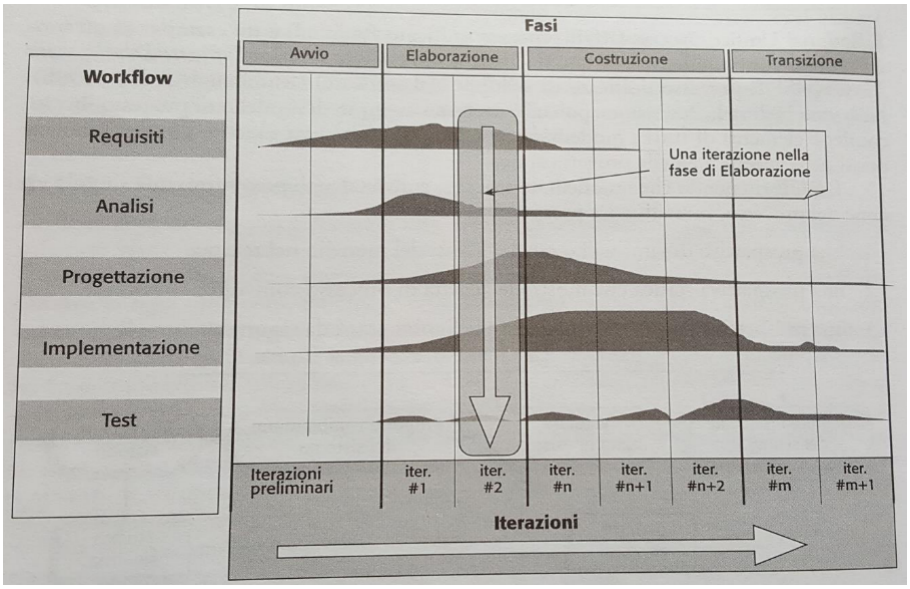
- **Modelli ibridi**: ogni sistema è pensato come insieme di sottosistemi, per ogni sistema è possibile adottare un diverso modello di sviluppo. Di norma conviene creare e raffinare prototipi funzionanti dell'intero sistema o di sue parti, secondo l'approccio incrementale-iterativo:

  1. Si individuano delle sottoparti relativamente autonome
  2. Si realizza il prototipo per una di esse
  3. Si continua con le altre parti
  4. Si aumenta il livello di dettagli di ogni parte iterativamente

- **Rational Unified Process**: modello di di sviluppo ibrido iterativo. Definisce uno schema generale adattabile a più situazioni anziché definire un schema preciso. Individua più prospettive:
    - **prospettiva dinamica**: mostra le fasi nel tempo
        1. Avvio: analisi di fattibilità calata nel contesto di un preciso mercato e di uno specifico caso d'uso (chi interagirà col sistema). Permette di ottenere una prima valutazione dei rischi e dei requisiti.
        2. Elaborazione: definisce la struttura complessiva del sistema, completa di analisi del dominio e di una prima fase di progettazione dell'architettura. Terminata questa fase sarà poi più difficile e costoso fare modifiche.
        3. Costruzione: progettazione e programmazione in parallelo delle varie parti del sistema. Questa fase comprende anche la verifica delle singole parti isolate e integrate.
        4. Transizione: fase in cui il sistema passa dall'ambiente di sviluppo a quello del cliente. Bisogna controllare che il sistema sia conforme alle specifiche del cliente e istruire gli utenti finali all'uso del prodotto.

           E' possibile che ci sia anche una retroazione tra 4 le fasi precedenti.
    - **Prospettiva statica**: si concentra sui workflow (attività di produzione). All'interno di una fase dinamica possono essere attivi anche tutti i workflow. RUP definisce 9 workflow (6 principali e 3 di supporto):
        - Modellazione delle attività aziendali (P)
        - Requisiti (P)
        - Analisi e progetto (P)
        - Implementazione (P)
        - Test (P)
        - Rilascio (P)
        - Gestione della configurazione e delle modifiche (S)
        - Gestione del progetto (S)
        - Ambiente (S)

          La prospettiva RUP è orientata ai modelli UML.

    - **Prospettiva pratica**: descrive delle buone prassi da usare nell'ingegneria del software:
        - Sviluppare software ciclicamente: pianificare gli incrementi e sviluppare le funzioni secondo la loro priorità
        - Gestire i requisiti: documentare esplicitamente i requisiti ed i cambiamenti effettuati analizzando l'impatto che i cambiamenti potrebbero avere

- Usare architetture basate sui componenti
- Creare modelli visivi del software
- Verificare la qualità del prodotto
- Controllare le modifiche del software con strumenti adatti

Analisi dei Requisiti

Requisiti di sistema

I requisiti di un sistema rappresentano la descrizione dei servizi forniti e dei vincoli operativi. I requisiti possono riguardare l'utente, ovvero dichiarare quali servizi il sistema dovrà fornire e sotto quali vincoli dovrà operare (generalmente sono posti in maniera astratta in linguaggio naturale), oppure riguardare il sistema, ovvero definiscono le funzioni, i servizi e i vincoli operativi del sistema in modo dettagliato (descritti all'interno di un documento apposito in maniera precisa e dettagliata).

I requisiti di sistema possono essere suddivisi in tre tipologie:

- **Requisiti funzionali**: descrivono ciò che il sistema dovrebbe fare elencando una serie di servizi. Per ogni servizio va indicato il comportamento del sistema sia in caso di input corretti che in caso si input errati e specificato cosa il sistema NON deve fare. Le specifiche dovrebbero essere complete e coerenti tra di loro.
- **Requisiti non funzionali**: non riguardano direttamente le funzionalità del prodotto, ma specificano dei vincoli aggiuntivi. In generale questi requisiti sono difficili da verificare perché espressi in modo vago e a volte sono in contrasto con i requisiti funzionali. Possono essere di più tipi:
    - **Requisiti del prodotto**: definiscono le proprietà del sistema (affidabilità, prestazioni, protezione, ...)
    - **Requisiti organizzativi**: riguardano il processo di sviluppo adottato
    - **Requisiti esterni**: identificano tutti i requisiti che derivano da fattori non provenienti dal sistema e dal suo processo di sviluppo (interoperabilità con sistemi esterni, requisiti legislativi, ...)
- **Requisiti di dominio**: Descrivono al progettista il dominio dell'applicazione e solitamente comportano l'utilizzo di una terminologia propria del dominio (es. il committente o chi di competenza deve specificare bene al progettista l'ambito di lavoro perché potrebbe non conoscerlo).

Analisi dei requisiti

L'analisi dei requisiti è necessaria per definire le proprietà che il sistema dovrà avere senza specificare una possibile implementazione. È il risultato di varie interazioni con l'utente per comprenderne i bisogni.

1. **Raccolta dei requisiti**: ha lo scopo di raccogliere tutte le informazioni sulle caratteristiche del sistema. Non prevede passi formali, ma prevede generalmente l'uso di interviste, osservazioni passive o studio di sistemi software già esistenti. Ha come risultato la redazione di un *documento testuale* (approvato dal cliente) e di una prima versione del *vocabolario* o *glossario*. In generale la gestione delle interviste è complessa in quanto i clienti potrebbero cambiare idea più volte, non essere in grado di spiegarsi oppure fornire requisiti in conflitto, fuori budget o irrealizzabili. Ogni requisito va validato (validità, consistenza, realizzabilità, completezza, controllo del budget...) e negoziato con il cliente prima di essere riportato nel *Documento dei Requisiti*. Tale documento specifica in modo chiaro, conciso ed univoco i requisiti (magari mediante una tabella) e deve contenere una prima versione del glossario. Il cambiamento dei requisiti è la norma, ma per ogni cambiamento va analizzato l'impatto che questo avrà sui tempi, sui costi e sulla fattibilità tecnica (cambiamenti tardivi sono più costosi e difficili da effettuare); per questo il sistema sviluppato deve essere il più possibile flessibile e robusto.

> *Dall'analisi dei sostantivi delle specifiche è possibile formulare un primo modello del dominio, mentre dall'analisi dei verbi un primo modello delle funzionalità. Ad ogni passo è poi necessario aggiornare la tabella dei requisiti procedendo fino a che ogni requisito non sia stato analizzato completamente.*

2. **Analisi del dominio**: ha l'obiettivo di definire la porzione del mondo reale rilevante per il sistema. E' di fondamentale importanza usare solo la terminologia di quello specifico dominio e dà come risultato una prima versione del vocabolario partendo dai sostantivi che si trovano nei requisiti.
3. **Analisi dei Requisiti**: ha l'obiettivo di definire il comportamento del sistema producendo un modello comportamentale (o dinamico) che ne descriva le funzionalità ad alto livello (si utilizza il **modello dei casi d'uso e scenari**). Per far questo è possibile usare:
    - *Scomposizione funzionale*: identificazione delle singole funzionalità del sistema. Si presenta il problema della mutabilità dei requisiti funzionali.
    - *Astrazione procedurale*: si considera ogni operazione come una singola entità a livello astratto
4. **Analisi dei Rischi**: occorre analizzare rischi relativi ai requisiti (sono completamente noti?), alle risorse umane, alla privacy ed alla protezione dei dati (vedi GDPR), alle tecnologie da usare, a fattori politici. La migliore strategia da adottare è quella preventiva (meglio non agire in maniera reattiva).

Casi d'uso e scenari

Permettono di formalizzare graficamente i requisiti formali e di comprendere al meglio il comportamento del sistema.

1. Individuazione dei confini del sistema
2. Individuazione degli **attori** (ruolo interpretato da un utente nei confronti del sistema) o **eventi** che rappresentano un qualsiasi avvenimento all'interno del sistema
3. Individuazione dei **casi d'uso** (servizio offerto dal sistema). Ogni caso d'uso viene sempre avviato direttamente o indirettamente da un attore che si pone un dato obiettivo e può concludersi con successo o fallendo.
4. Disegno del **diagramma dei casi d'uso** (mette in risalto le associazioni tra attori e casi d'uso)
5. Descrizione dei dettagli dei singoli casi d'uso dal punto di vista degli attori. Essa comprende sempre:
    - 0+ precondizioni : condizioni che devono essere tutte verificate prima che il caso d'uso abbia inizio
    - 1+ scenari (di cui uno principale ed eventuali alternativi mediante ramificazioni e iterazioni) : descrizione attraverso una sequenza di passi scritta in forma narrativa strutturata e utilizzando il vocabolario di dominio.
    - 0+ postcondizioni : condizioni che sono essere tutte verificate se il caso d'uso termina con successo

    Per relazionare i casi d'uso è possibile usare le tecniche di:

    - generalizzazione/specializzazione : quando un caso d'uso è simile ad un altro ma fa qualcosa di più (può essere utilizzato anche per gli attori)

      (freccia piena con punta chiusa)
    - inclusione : quando un caso d'uso "usa" almeno una volta un altro

      (freccia tratteggiata con clausola «include»)
    - estensione : si utilizza quando è necessario aggiungere un comportamento opzionale ad un caso d'uso esistente

      (freccia tratteggiata con clausola «extend»)
6. Validazione (con firma) del grafico con il cliente

Sicurezza e privacy (GDPR)

Sicurezza e privacy vanno progettati assieme al sistema, con un *livello minimo di default*, non possono essere aggiunte successivamente (non è sufficiente cifrare i file). Il tipo di protezione dipende dal valore, dalla dimensione e dal tempo di vita delle informazioni (questo perché la protezione ha un costo).

General Data Protection Regulation (GDPR) sostituisce la Data Protection Directive dal 2018 e stabilisce l'obbligo di aderenza per tutti i prodotti che trattano dati personali.

Pseudonimizzazione: processo di trattamento dei dati che rendere possibile il collegamento tra dei dati ad un interessato solo in caso si sia in possesso di alcune informazioni aggiuntive conservate in separata sede.

La normativa prevede che:

- L'utente deve essere informato dell'uso di certi dati e tali dati devono essere usati solo per lo scopo per cui erano stati raccolti.
- Vanno richiesti solo i dati pertinenti al dominio in questione
- Le modifiche ai contratti e ai dati vanno immediatamente notificate
- I dati vanno conservati solo per il periodo in cui sono necessari, non oltre
- I dati vanno conservati con sicurezza adeguata
- Non deve essere possibile che i dati siano resi accessibili ad un numero indefinito di persone fisiche senza il consenso della persona fisica a cui i dati fanno riferimento
- Va garantito un livello di protezione adeguato al caso d'uso
- Va tenuto conto del fatto che gli operatori sono umani (È inutile usare tante password lunghissime, il risultato sarà un post-it sulla scrivania)
- Deve essere possibile eliminare tutti i dati relativi ad un utente su richiesta di quell'utente (anche quelli contenuti in tutti i backup delle basi di dati)

Sicurezza informatica

Solitamente l'obiettivo degli attacchi è la ricezione di dati in maniera illecita, perciò obiettivo della sicurezza informatica è la protezione da potenziali rischi e/o violazioni dei dati

- **riservatezza**
- **integrità** e **autenticità**
- **disponibilità**

Per la protezione è necessario quindi **identificare**, **autenticare** ("qualcosa che si è, qualcosa che si ha, qualcosa che si sa") e **autorizzare** l'utenza che accede ai dati e deve essere garantito un funzionamento affidabile (es. resistenza ad attacchi DoS).

Nella scelta della sicurezza incidono diversi fattori economici e ingegneristici:

- Dinamicità del sistema
- Dimensione del sistema
- Metodologie di accesso
- Tempo di vita delle informazioni
- Costi della protezione
- Costi in caso di violazione
- Valore percepito e tipologia di attaccante

In generale un sistema è forte tanto quanto il suo punto più debole (**Catena degli anelli**), e la sicurezza va progettata sia a livello fisico che logico (uso di chiavi per cifrare i dati, comunicazioni cifrate, messaggi autenticati (firma digitale), controllo forte degli accessi, controllo forte dei permessi, ...).

Progettazione della sicurezza

I **Sistemi critici** sono sistemi tecnici o socio-tecnici da cui dipendono persone o servizi. Se questi sistemi non forniscono i loro servizi come ci si aspetta possono verificarsi seri problemi e importanti perdite. Ci sono tre tipi principali di sistemi critici:

- sistemi *safety-critical* i cui fallimenti causano perdite umane, incidenti o danni ambientali
- *mission-critical* i cui fallimenti causano problemi all'erogazione dei servizi
- *business-critical* i cui fallimenti causano problemi a delle aziende

Il sistema deve essere *disponibile* ossia deve poter erogare il suo servizio sempre, *affidabile* ossia dare gli stessi risultati in caso vengano sottoposti allo stesso stimolo, *sicuro* ossia garantire una certa sicurezza nell'uso. È necessario tenere conto delle varie cause dei fallimenti *hardware*, *software* o *comportamento umano*.

Nei confronti dei sistemi critici la rete è ambivalente: da un lato permette di distribuire più efficacemente le patch di sicurezza e di divulgare vulnerabilità, ma dall'altro rappresenta una modalità d'accesso al sistema difficile da disciplinare.

Esempi di possibili attacchi sono:

- *exploit* : metodo che sfrutta un bug o una vulnerabilità, per l'acquisizione di privilegi
- *buffer overflow* : fornire al programma più dati di quanto esso si aspetti di ricevere, in modo che una parte di questi vadano scritti in zone di memoria dove sono, o dovrebbero essere, altri dati o lo stack del programma stesso
- *shell code* : sequenza di caratteri che rappresenta un codice binario in grado di lanciare una shell, può essere utilizzato per acquisire un accesso alla linea di comando
- *sniffing* : attività di intercettazione passiva dei dati che transitano in una rete
- *cracking* : modifica di un software per rimuovere la protezione dalla copia, oppure per ottenere accesso ad un'area riservata
- *spoofing* : tecnica con la quale si simula un indirizzo IP privato da una rete pubblica facendo credere agli host che l'IP della macchina server da contattare sia il suo
- *trojan* : programma che contiene funzionalità malevole
- *DoS* : il sistema viene forzatamente messo in uno stato in cui i suoi servizi non sono disponibili, influenzando così la disponibilità del sistema

Nella progettazione di un sistema informatico non si può prescindere dalla componente relativa alla sicurezza, ma il sistema va progettato tenendo conto anche delle minacce a cui andrà in contro. Si parla quindi di **Security engineering** e comprende ad esempio la *gestione degli utenti e di permessi* per quanto riguarda la prevenzione di usi illeciti di risorse; l'*installazione, la configurazione e l'aggiornamento del software* e *monitoraggio degli attacchi, rilevazione e ripristino.*

Sicurezza: Glossario

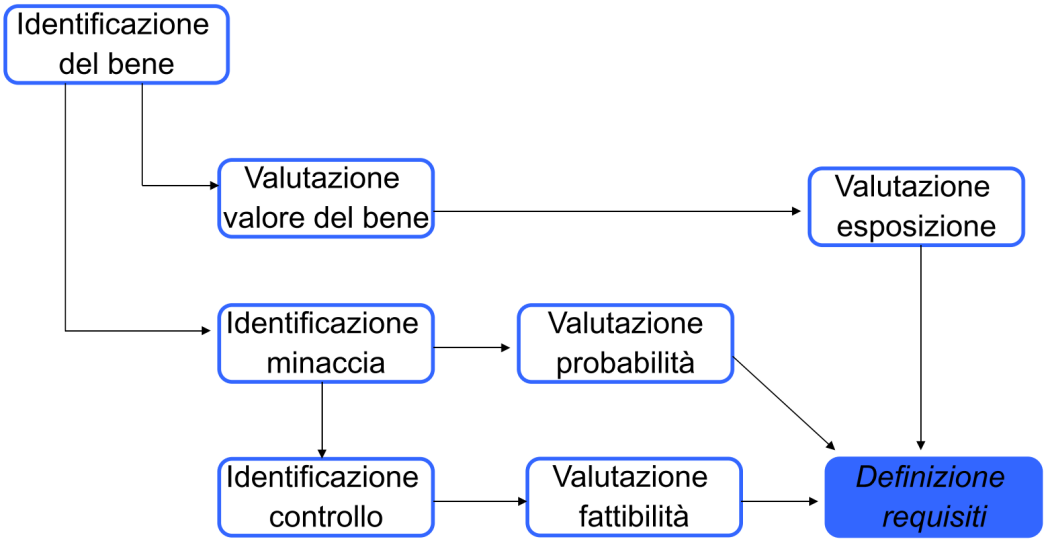
- *Bene (Asset)*: una risorsa del sistema che deve essere protetta
- *Esposizione (Exposure)*: possibile perdita o danneggiamento come risultato di un attacco andato a buon fine. Potrebbe essere una perdita o un danneggiamento di dati o una perdita di tempo nel ripristino del sistema dopo l'attacco
- *Vulnerabilità (Vulnerability):* una debolezza nel sistema software che potrebbe essere sfruttata per causare una perdita o un danno
- *Attacco (Attack)*: sfruttamento di una vulnerabilità del sistema
- *Minaccia (Threat):* circostanza che ha le potenzialità per causare perdite e danni
- *Controllo (Control):* una misura protettiva che riduce una vulnerabilità del sistema

Analisi del Rischio

L'**analisi del rischio** si occupa della valutazione delle perdite che un attacco può causare e del bilanciamento dei costi relativi alla protezione. Infatti l'ammontare dei costi deve sempre essere inferiore all'ammontare dei danni provocati da un possibile attacco. È suddivisa in una prima fase di valutazione preliminare e successivamente in un ciclo di analisi contestuale alla progettazione del sistema.

Valutazione preliminare

Questa fase permette di avere una prima idea di quali saranno le risorse da proteggere (*beni*) e di quali saranno le minacce ed i costi associati.



1. **Identificazione dei beni**: in questa fase vengono analizzate le varie risorse che compongono il sistema. In prima battuta il sistema viene visto come un insieme di *risorse fisiche* (ossia come un insieme di dispositivi che hanno bisogno di alimentazione e locali adeguati) da analizzare. Successivamente si passa ad una vista logica. Il sistema è visto come un insieme di *risorse logiche* (quali informazioni e flussi di informazioni). In questa fase occorre identificare e classificare sia le informazioni sia i servizi che il sistema eroga in base al valore che rivestono per l'organizzazione che le gestisce. In ultimo si analizzano le *dipendenze tra risorse*. Per ogni risorsa (logica o fisica che sia) si identifica l'albero delle dipendenze al fine di trovare quali siano le *risorse potenzialmente critiche*, ossia quelle da cui dipende il funzionamento di un alto numero di altre risorse.
2. **Identificazione delle minacce**: in questa fase si definisce quello che non deve accadere al sistema (inizialmente considerando come evento indesiderato ogni accesso non esplicitamente permesso), identificando e separando gli *eventi accidentali* dagli *attacchi intenzionali*. Gli attacchi intenzionali possono essere distinti in base all'obiettivo in attacchi a *livello logico* (quegli attacchi che mirano a sottrarre informazioni dal sistema o a degradarne l'operatività) e in attacchi a *livello fisico* (tesi a sottrarre o danneggiare risorse critiche). Gli eventi accidentali invece rappresentano tutta quella serie di eventi che non dipendono dall'azione diretta di un attaccante ma unicamente dall'usura dei dispositivi. Possono riguardare sia il livello fisico (guasti ai dispositivi), sia il livello logico (perdita di password, cancellazione accidentali di file, corruzione del sistema...).
3. **Valutazione dell'esposizione**: ad ogni minaccia viene associato un rischio, inteso come la combinazione della probabilità che un evento accada e dei danni che tale evento può arrecare. Nell'eseguire questa valutazione si tiene conto anche della dipendenza tra le risorse per identificare una eventuale propagazione del malfunzionamento.
4. **Valutazione delle probabilità**. Per quanto riguarda gli attacchi intenzionali tale valore dipende sia dal valore portato dalla riuscita dell'attacco sia dalla facilità con cui l'attacco può essere portato a termine. Va tenuto conto inoltre del fatto che un attaccante potrebbe usare un attacco composto per perseguire i propri obiettivi. Per quanto riguarda invece gli eventi accidentali la probabilità di malfunzionamento è dettata dai dati forniti dal costruttore e dall'uso più o meno proprio della risorsa stessa.
5. **Identificazione del controllo**. Una volta identificate le minacce occorre capire come neutralizzarle. Occorre sempre tener presente il rapporto *costo/efficacia* ed il *grado di adeguatezza* in modo da evitare controlli che implichino dei costi ingiustificati e valutare i *costi nascosti* dati ad esempio dalla messa in opera dei controlli o dal decadimento delle prestazioni del sistema. A livello organizzativo la protezione migliore è rappresentata dalla consapevolezza degli utenti rispetto all'uso che fanno dei sistemi e delle interfacce. Questo uso deve essere disciplinato da un sistema più o meno forte di controllo dei permessi e dalla definizione di utenti e ruoli, per ognuno dei quali l'amministratore dovrà poi definire un insieme di norme e di comportamenti da seguire. A livello tecnico i controlli sono presenti sia a livello di sistema operativo che specifici per ogni particolare sistema, mediante, ad esempio, la compartimentazione delle informazioni e delle applicazioni. Gli insiemi dei controlli non devono essere una collezione disorganizzata, ma come un

insieme *completo*, *omogeneo*, *attuabile* di controlli (eventualmente ridondanti in maniera controllata).

Ciclo di vita della valutazione del rischio

Viene effettuato durante tutto il ciclo di vita del software. Infatti dalla progettazione o da sistemi preesistenti possono essere ereditate delle vulnerabilità, magari anche nascoste o difficili da trovare, pertanto esse vanno identificate e va valutata una contromisura adatta.

Security Use Case e Misuse Case

I **misuse case** si concentrano sulle interazioni tra l'applicazione e gli attaccanti che cercano di violarla, e hanno come condizione di successo un attacco andato a buon fine. Sono utili per l'analisi delle minacce ma non per l'identificazione dei requisiti di sicurezza.

Compito dei **security use case** è specificare i requisiti tramite i quali l'applicazione dovrebbe essere in grado di proteggersi dalle minacce specificate all'interno dei misuse case.

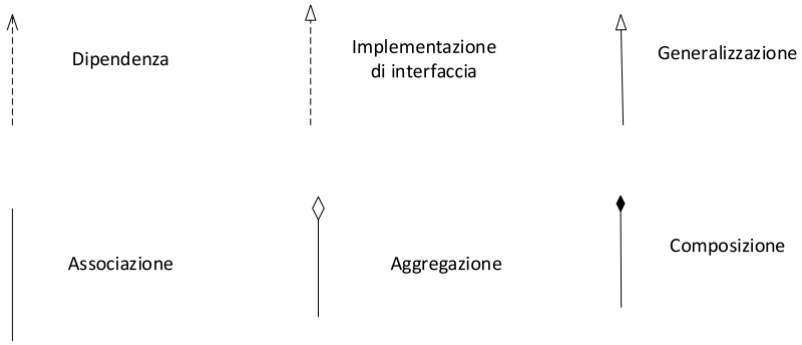
Requisiti di sicurezza

L'approccio convenzionale della specifica dei requisiti, spesso specificati nella forma "non deve", è basato sul contesto, sui beni da proteggere e sul loro valore per l'organizzazione. Si identificano varie categorie di requisiti per la sicurezza:

- **Requisiti di identificazione**: specificano se un sistema deve identificare gli utenti prima di interagire con loro
- **Requisiti di autenticazione**: specificano come identificare gli utenti
- **Requisiti di autorizzazione**: specificano i privilegi e i permessi di accesso degli utenti identificati
- **Requisiti di immunità**: specificano come il sistema deve proteggersi da virus, worm e minacce simili
- **Requisiti di integrità**: specificano come evitare la corruzione dei dati
- **Requisiti di scoperta delle intrusioni**: specificano quali meccanismi utilizzare per scoprire gli attacchi al sistema
- **Requisiti di non-ripudiazione**: specificano che una parte interessata in una transazione non può negare il proprio coinvolgimento
- **Requisiti di riservatezza**: specificano come deve essere mantenuta la riservatezza delle informazioni
- **Requisiti di controllo della protezione**: specificano come può essere controllato e verificato l'uso del sistema
- **Requisiti di protezione della manutenzione del sistema**: specificano come una applicazione può evitare modifiche autorizzate da un accidentale annullamento dei meccanismi di protezione

Diagrammi UML

UML è un linguaggio di modellazione (Unified Modeling Language) che serve per visualizzare, specificare, costruire, documentare un sistema e gli elaborati prodotti durante il suo sviluppo dotato di una sintassi e di una semantica standard basate su un metamodello integrato ma è estensibile e personalizzabile.



Dipendenza

Una dipendenza indica che cambiamenti dell'elemento indipendente influenzano l'elemento dipendente. Si rappresenta con una freccia tratteggiata con punta aperta. UML mette a disposizione nove diversi tipi di dipendenza, ma per i nostri fini consideriamo quasi sempre «use»

Package

Un package è utilizzato per raggruppare elementi e fornire loro uno spazio di nomi. Ogni package può essere innestato in altri package a sua volta.

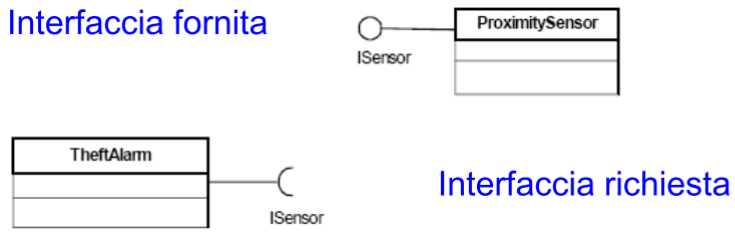
Diagramma dei package

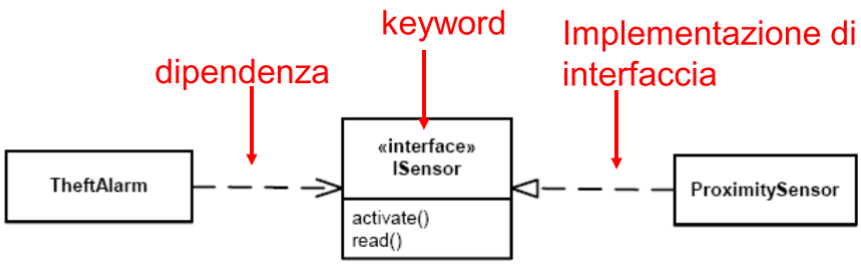
I diagrammi dei package illustrano come gli elementi della modellazione siano organizzati a livello di package, specificando le dipendenze (dipendenze logiche, a livello di package e non di contenuto).

> *Non è detto che le dipendenze logiche rimangano tali anche nella fase di progettazione.*

Interfaccia

Le interfacce forniscono un modo per partizionare e caratterizzare gruppi di proprietà. Un'interfaccia non deve specificare come possa essere implementata, ma semplicemente quello che è necessario per poterla realizzare. Un'interfaccia è rappresentata dalla parola chiave «interface».





Classe astratta

Una classe astratta è una classe che non può essere direttamente istanziata: per farlo bisogna prima crearne una sottoclasse concreta. Tipicamente, una classe astratta ha una o più operazioni astratte cioè operazioni pubbliche che non hanno implementazione.

> *Una classe o un'operazione astratta viene generalmente scritta in corsivo.*

Classe

Una classe modella un insieme di entità aventi tutti lo stesso tipo di caratteristiche (attributi, associazioni, operazioni...)

Enumerazione

Le enumerazioni sono usate per mostrare un insieme di valori prefissati che non hanno altre proprietà oltre al loro valore simbolico e sono rappresentate dalla parola chiave «enumeration».

> *Le operazioni statiche vanno sottolineate sul diagramma.*

Diagramma delle classi

Questo diagramma descrive quali siano le entità, ognuna descritta con le proprie **proprietà** e **operazioni**, facenti parte del sistema e i **vincoli** che si applicano alla classi e alle **relazioni** che sussistono tra di esse.

Proprietà

Le proprietà rappresentano le caratteristiche strutturali di una classe e sono:

- **Attributi**

  La notazione degli attributi descrive una proprietà con una riga di testo all'interno del box della classe. La forma completa è:

  ```
  visibilità nome : tipo molteplicità = default {stringa di proprietà}
  ```

  La **visibilità** può essere: pubblico (`+`), privato (`-`), package (`~`) o protected (`#`).

  La **molteplicità** è l'indicazione di quanti oggetti possono entrare a far parte di una proprietà e può essere `1`, `N..M`, `*`.

  > *Gli attributi statici vanno sottolineati sul diagramma.*

- **Associazioni**

  Sono un modo differente di rappresentare delle proprietà. Ogni associazione è data da una linea continua che unisce due classi ed ha un **nome**. La **molteplicità** va indicata alle estremità della linea di congiunzione dove possiamo assegnare dei **ruoli** che indicano la partecipazione delle entità all'associazione. Nel caso di **associazioni ternarie** le linee che partono dalle entità confluiscono in un simbolo **diamante** posto al centro.

  Alcuni esempi notevoli sono:

  - **Aggregazione** (disegnata con un rombo vuoto dal lato del contenitore): rappresenta una relazione *intero-parte*. Le parti in questo caso hanno una vita indipendente da quella dell'intero, pertanto sopravvivono alla sua distruzione.
  - **Composizione** (disegnata con un rombo pieno dal lato del contenitore): oltre ai vincoli posseduti dall'aggregazione ve ne sono altri:
    - una parte può essere inclusa da un solo intero per volta
    - la distruzione dell'interno implica la distruzione delle parti
  - **Generalizzazione** (freccia con la punta chiusa e vuota): rappresenta una relazione di tipo sottoclasse-superclasse. Indica che ogni istanza della sottoclasse è anche istanza della superclasse. A seconda della gerarchia delle classi questa relazione può essere *disgiunta/sovrapposta* e *completa/incompleta*. In una generalizzazione la sottoclasse non solo può avere caratteristiche aggiuntive rispetto alla superclasse, ma può anche sovrascrivere (overriding) le proprietà ereditate dalla superclasse. Inoltre può essere *singola/multipla* a seconda se un oggetto appartiene a un solo tipo o è descritto da più tipi non necessariamente collegati dall'ereditarietà.

Operazioni

Le operazioni sono le azioni che la classe sa eseguire, e in genere si fanno corrispondere direttamente ai metodi della corrispondente classe a livello implementativo. La forma completa è:

```
visibilità nome( lista parametri ) : tipo ritorno {stringa di proprietà}
```

La **visibilità** può essere: pubblico (`+`), privato (`-`), package (`~`) o protected (`#`).

Diagrammi di sequenza

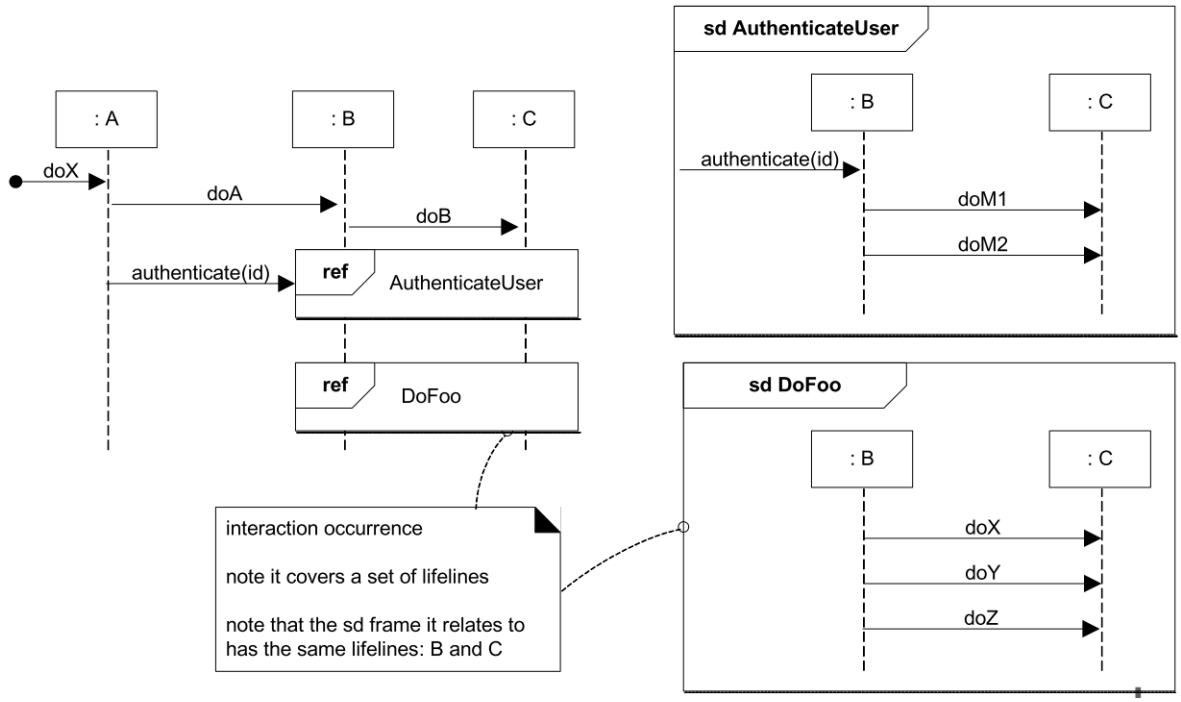
Sono diagrammi che illustrano le interazioni tra le classi/entità disponendole lungo una sequenza temporale. Mostra i soggetti (chiamati tecnicamente **lifeline**) che partecipano all'interazione e la sequenza dei messaggi scambiati tra di essi. Nei diagrammi troviamo in ascissa i diversi soggetti (anche non in ordine di esecuzione), mentre in ordinata abbiamo la scala dei tempi sviluppata

verso il basso.

L'ordine con cui le gli eventi si verificano nelle lifeline rappresenta esattamente l'ordine in cui tali eventi si devono verificare. La distanza in termini grafici tra due eventi non ha rilevanza dal punto di vista semantico.

Dal punto di vista notazionale, una lifeline è rappresentata da un rettangolo che costituisce la "testa" seguito da una linea verticale tratteggiata che rappresenta il tempo di vita del partecipante.

Spesso i diagrammi di sequenza possono assumere una certa complessità quindi si ha la necessità di definire comportamenti più articolati come composizione di nuclei di interazione più semplici oppure potrebbe essere utile definire un'operazione frequente una sola volta e richiamarla dove necessario. Per questa ragione UML permette di inserire riferimenti ad altri diagrammi detti **InteractionUse**. I punti di interconnessione tra i due diagrammi prendono il nome di **gate**. Esempio:



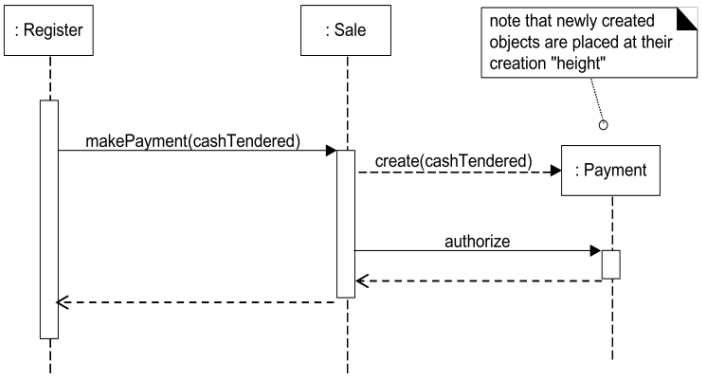
Le lifeline si scambiano dei messaggi che possono essere *sincroni* (da cui ci si aspetta un qualche oggetto ritornato) o *asincroni*.

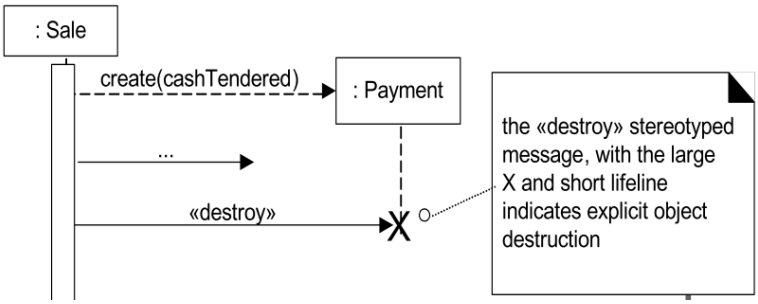
Si distinguono:

- *complete message* (freccia nomale) se specificati sia mittente che destinatario
- *lost message* (freccia con pallino pieno sul destinatario) se specificato solo il mittente
- *found message* (freccia con pallino pieno sul mittente) se specificato solo il destinatario
- *unknown message* (freccia con pallino pieno su mittente e destinatario) se non sono specificati né mittente né destinatario

Anche le frecce hanno significati differenti:

- riga continua freccia piena: messaggio di call sincrono
- riga continua freccia vuota: messaggio di call asincrono
- riga tratteggiata freccia vuota. indica il ritorno di un messaggio

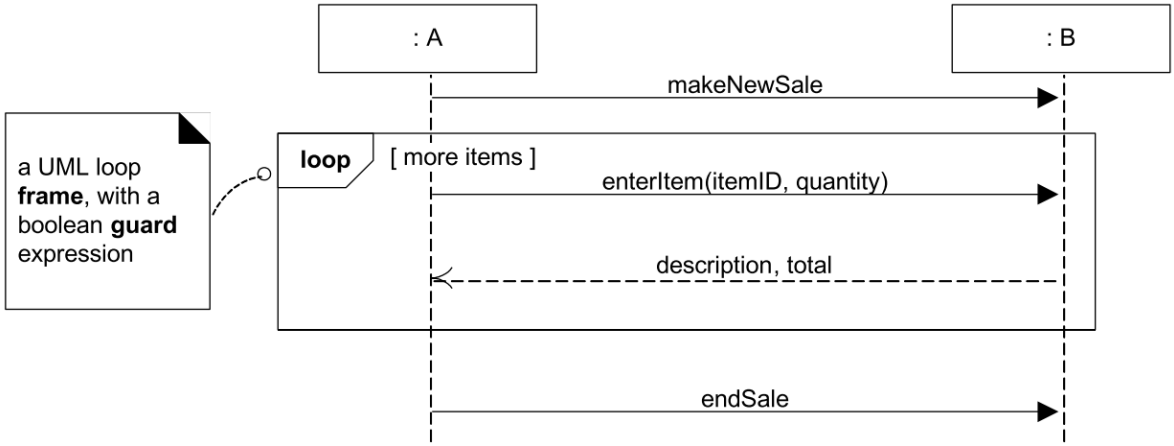




È possibile esprimere comportamenti più complessi tramite i **combined fragment**, che permettono di descrivere catene di eventi:

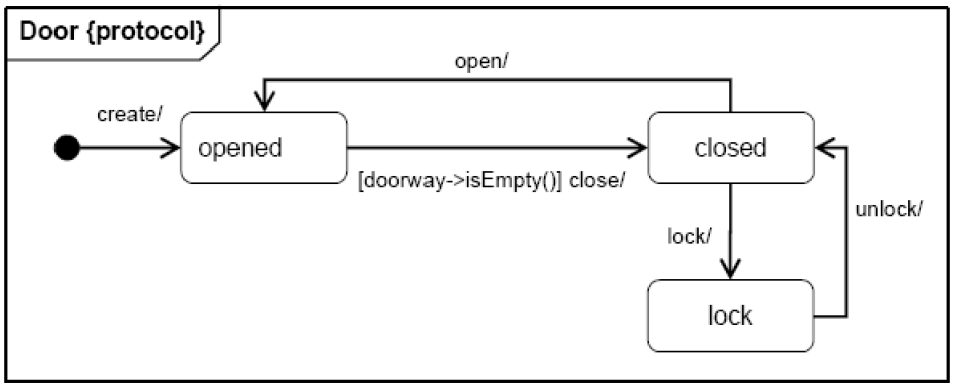
- Loop: ciò che è racchiuso nell'operando sarà eseguito ciclicamente finché la guardia sarà verificata
- Alternatives (alt): indica che sarà eseguito il contenuto di uno solo degli operandi, quello la cui guardia risulta verificata
- Optional (opt): indica che l'esecuzione del contenuto dell'operando sarà eseguita solo se la guardia è verificata
- Break (break): ha la stessa semantica di opt, con la differenza che in seguito l'interazione sarà terminata
- Critical: specifica un blocco di esecuzione atomico (non interrompibile)
- Parallel (par): specifica che il contenuto del primo operando può essere eseguito in parallelo a quello del secondo
- Weak Sequencing (seq): specifica che il risultato complessivo può essere una qualsiasi combinazione delle interazioni contenute negli operandi, purché:
    1. l'ordinamento stabilito in ciascun operando sia mantenuto nel complesso
    2. eventi che riguardano gli stessi destinatari devono rispettare anche l'ordine degli operandi, cioè i messaggi del primo operando hanno precedenza su quelli del secondo
    3. eventi che riguardano destinatari differenti non hanno vincoli di precedenza vicendevole
- Strict Sequencing (strict): indica che il contenuto deve essere eseguito nell'ordine in cui è specificato, anche rispetto agli operandi
- Ignore: indica che alcuni messaggi, importanti ai fini del funzionamento del sistema, non sono stati rappresentati, perché non utili ai fini della comprensione dell'interazione
- Consider: è complementare ad ignore
- Negative (neg): racchiude una sequenza di eventi che non deve mai verificarsi
- Assertion (assert): racchiude quella che è considerata l'unica sequenza di eventi valida. Di solito è associata all'utilizzo di uno State Invariant come rinforzo

Esempio:



Diagrammi di stato

I diagrammi di stato modellano la dipendenza che esiste tra lo stato di una classe/entità ed i messaggi/eventi che questo riceve in ingresso. Ne specifica il ciclo di vita definendo le regole che lo governano. Quando una classe/entità si trova in un certo stato può essere interessato a determinati eventi (e non ad altri, alcuni eventi possono essere ignorati o non avere effetto) e come risultato di un evento una classe / entità può passare ad un nuovo stato (transizione). Esempio:



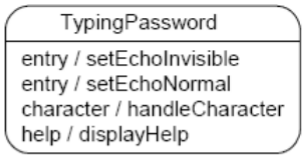
I concetti più importanti di un diagramma di stato sono:

- gli **stati**, indicati con rettangoli con angoli arrotondati
- le **transizioni** tra stati, indicati attraverso frecce
- gli **eventi** che causano transizioni, la tipologia più comune è rappresentata dalla ricezione di un messaggio, che si indicano semplicemente scrivendo il nome del messaggio con relativi argomenti vicino alla freccia
- i **marker** di inizio e fine rappresentati rispettivamente da un cerchio nero con una freccia che punto allo stato iniziale e come un cerchio nero racchiuso da un anello sottile
- le **azioni** che una entità è in grado di eseguire in risposta alla ricezione di un evento
- il **vertice** che rappresenta l'astrazione di nodo nel diagramma e può essere la sorgente o la destinazione di una o più transizioni

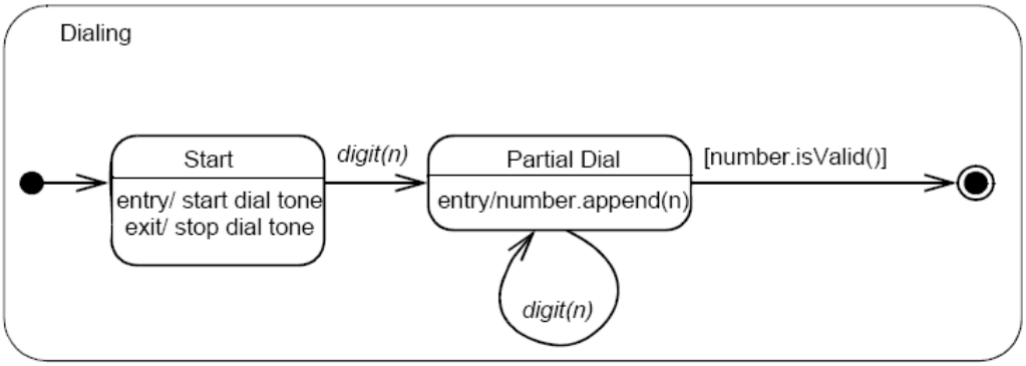
Stati

- **Stato semplice**: stato che non ha sottostati
- **Stato composto**: stato che ha sottostati, diviso in **regioni**. Ogni regione si comporta come un diagramma degli stati a sé stante. Tutte le regioni vengono completate in ordine, uno stato composto si può dire completato quando tutte le regioni ortogonali hanno completato il proprio comportamento
- **Stato submachine**: rappresenta il diagramma di stato di una sottoparte del sistema

Ogni stato al suo interno può specificare delle azioni da eseguire al momento di ingresso dello stato, al momento di transizione in uscita, oppure durante la permanenza dall'interno dello stato.

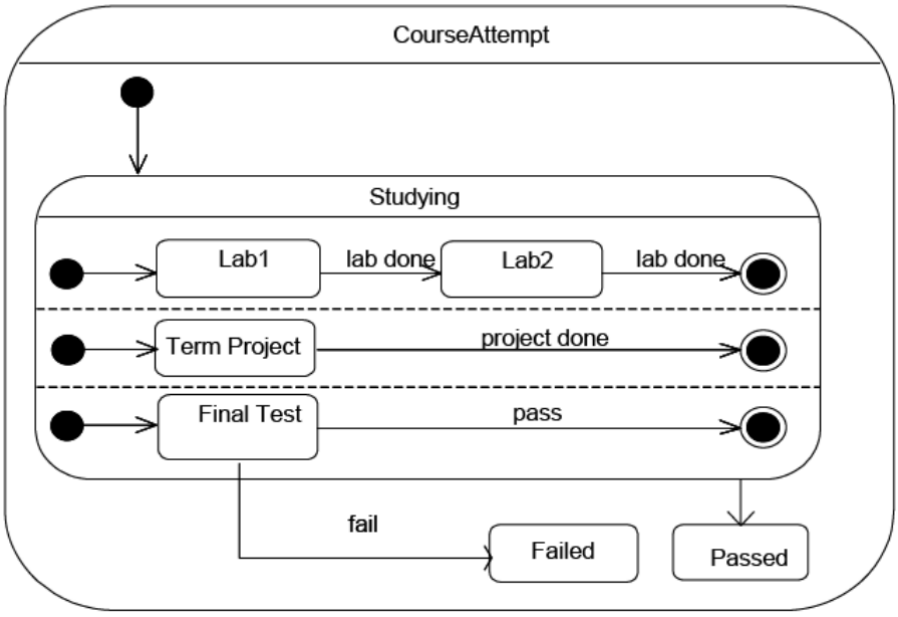


Simple state con azioni



Composite state

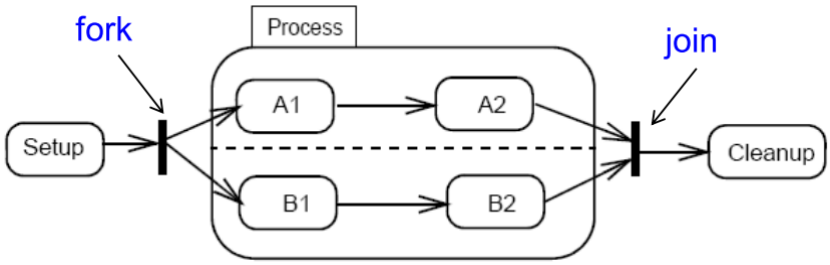
Composite state con regioni ortogonali



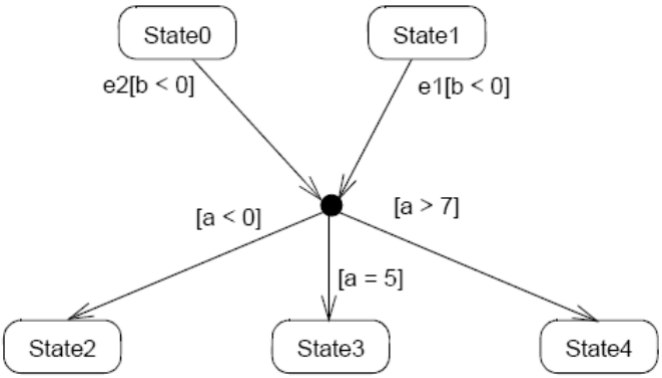
Pseudostati

- **Initial**: vertice di default che è la sorgente della singola transizione verso lo stato di default di un composite state. Ci può essere al massimo un vertice initial e la transizione uscente da questo vertice non può avere trigger o guardie
- **deepHistory**: la più recente configurazione attiva del composite state che contiene direttamente questo pseudostato
- **shallowHistory**: il più recente substate attivo di un composite

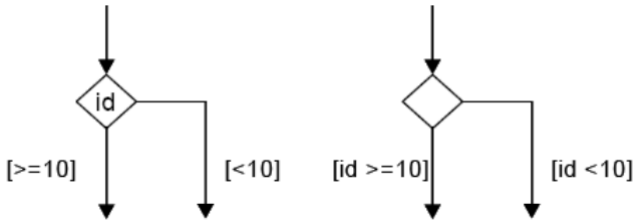
- **Join**: permette di eseguire il merge di diverse transizioni provenienti da diverse sorgenti appartenenti a differenti regioni ortogonali.
- **Fork**: permette di separare una transizione entrante in due o più transizioni che terminano in vertici di regioni ortogonali.



- **Junction**: è un vertice privo di semantica che viene usato per "incatenare" insieme transizioni multiple.
  È usato per costruire percorsi di transizione composti tra stati:



- **Choice**: è un tipo di vertice che quando viene raggiunto causa la valutazione dinamica delle **guardie dei trigger** delle transizioni uscenti. Le guardie sono quindi tipicamente scritte sotto forma di "funzione" che viene valutata al momento del raggiungimento del vertice choice. Permette di realizzare un branch condizionale dinamico:

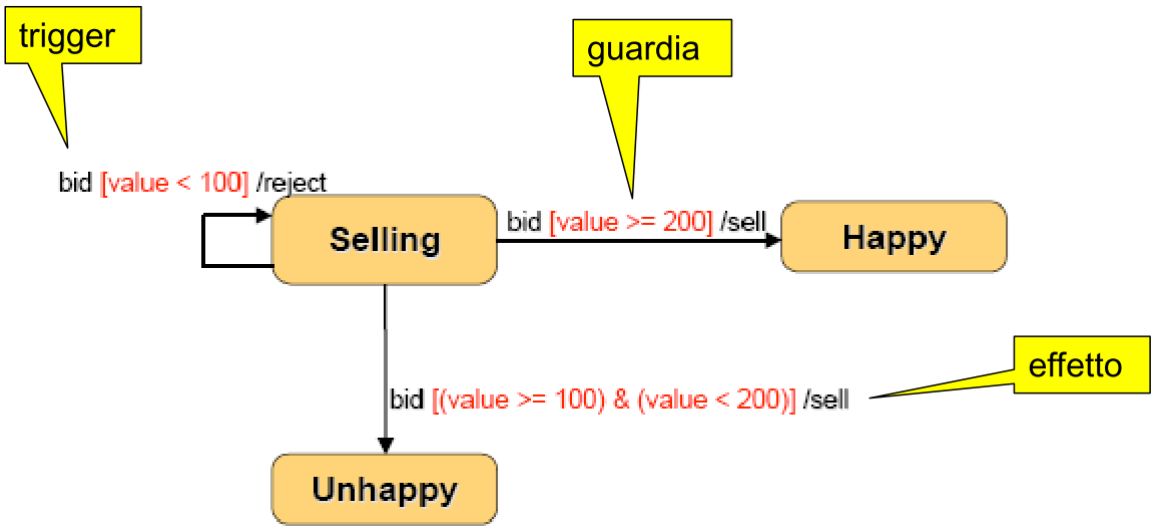


- **Entry point**: è l'ingresso di uno state machine o di un composite state
- **Exit point**: è l'uscita da uno state machine o da uno stato composito. Entrare in questo vertice significa triggerare la transizione che ha questo vertice come sorgente
- **Terminate**: entrare in questo vertice implica che l'esecuzione di questo state machine è terminata. Lo state machine non uscirà da nessuno stato né verranno invocate altre azioni a parte quelle associate con la transizione che porta allo stato terminate

Transizioni

Per passare da uno stato all'altro si sfruttano le transizioni, ovvero delle relazioni dirette tra due stati. Per ogni transizione è specificato l'evento che la scatena con una guardia (una specifica più fine sulle condizioni che scatenano la transizione):

- **trigger**: cioè i tipi di evento che possono innescare la transizione
- **guardia**: cioè un vincolo che fornisce un controllo fine sull'innesco della transizione. La guardia è valutata quando una occorrenza dell'evento è consegnata allo stato. Se la guardia risulta verificata allora la transizione può essere abilitata altrimenti questa viene disabilitata.
- **effetto**: specifica un comportamento opzionale che deve essere eseguito quando la transizione scatta



Eventi

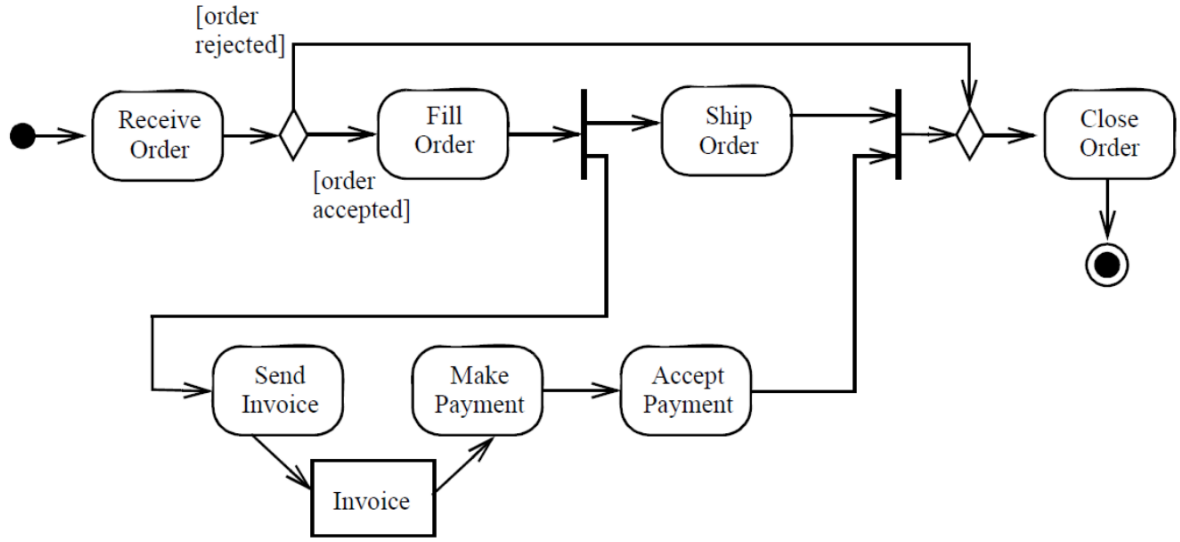
- Evento di chiamata: ricezione di un messaggio che richiede l'esecuzione di una operazione
- Evento di cambiamento: si verifica quando una condizione passa da "falsa" a "vera". Tale evento si specifica scrivendo «when» seguito da un'espressione, racchiusa tra parentesi tonde, che descrive la condizione. Utile per descrivere la situazione in cui un oggetto cambia stato perché il valore dei suoi attributi è modificato dalla risposta ad un messaggio inviato
- Evento segnale: consiste nella ricezione di un segnale
- Evento temporale: espressione che denota un lasso di tempo che deve trascorrere dopo un evento dotato di nome. Con «after» si può far riferimento all'istante in cui l'oggetto è entrato nello stato corrente, mentre con «at» si esprime qualcosa che deve accadere in un particolare momento

Azione

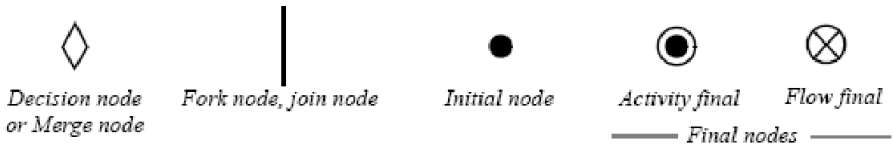
- Entry: tutte le volte che si entra in uno stato viene generato un evento di entrata a cui può essere associato uno o più specifici comportamenti che vengono eseguiti prima che qualsiasi altra azione possa essere eseguita
- Exit: tutte le volte che si esce da uno stato viene generato un evento di uscita a cui può essere associato uno o più specifici comportamenti che vengono eseguiti come ultimo passo prima che lo stato venga lasciato
- Do: rappresenta il comportamento che viene eseguito all'interno dello stato

Diagramma delle attività

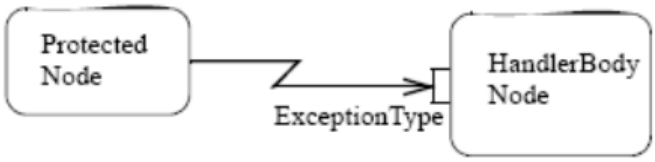
I diagrammi delle attività descrivono il modo in cui diverse attività sono coordinate e possono essere usati per mostrare l'implementazione di una operazione. Mostrano le attività di un sistema in generale e delle sottoparti, specialmente quando un sistema ha diversi obiettivi e si desidera modellare le dipendenze tra essi prima di decidere l'ordine in cui svolgere le azioni. I diagrammi delle attività sono utili anche per descrivere lo svolgimento dei singoli casi d'uso e la loro eventuale dipendenza da altri casi.



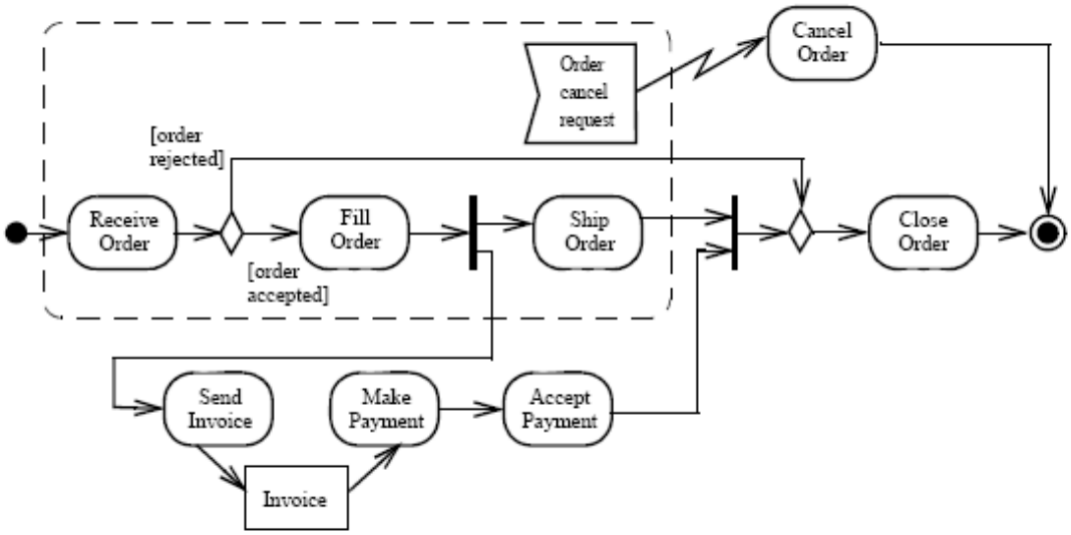
Ogni **attività** indica una porzione di lavoro che deve essere svolto e può essere collegata ad altre attività tramite degli **archi** di transizione. È possibile specificare dei **punti di decisione** (aventi una transizione entrante e più transizioni uscenti, una sola delle quali potrà essere scelta), **fork** (determina l'esecuzione in parallelo di più attività), **merge/join** (rappresentano il punto in cui più transizioni si uniscono).



Exception handler

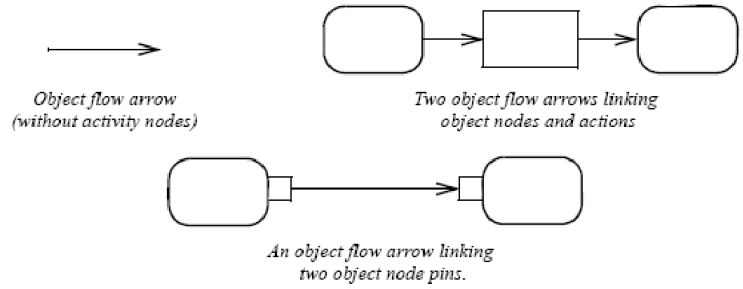


Interruptable activity region



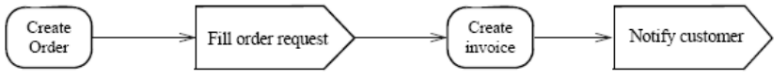
Object flow

È un arco che ha oggetti o dati che fluiscono su di esso:

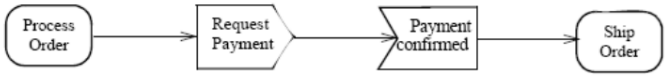


Notazione

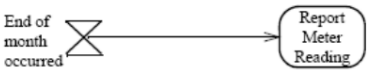
Segnali ed eventi



SendSignal Action



AcceptSignal Action



Evento ripetuto nel tempo

Diagramma dei componenti

Specifica un contratto formale di servizi offerti e richiesti in termini di interfacce (eventualmente esposte tramite porte). Il concetto di componente è strettamente legato a quello di struttura composita che spesso viene impiegata per rappresentare le parti interne del componente. Un componente è tipicamente specificato da uno o più classificatori (ad es. classi) e può essere implementato da uno o più artefatti (file eseguibile, script, ...).

l Diagramma di Struttura Composita ha l'obiettivo di rappresentare la struttura interna (le parti) di un classificatore (classe, componente...), inclusi i punti di interazione (porte) utilizzati per accedere alle caratteristiche della struttura.

Diagramma di deployment

Tali diagrammi di deployment documentano la distribuzione fisica di un sistema, mostrando su quali macchine fisiche i vari componenti debbano essere eseguiti. Mostrano i collegamenti che permettono la comunicazione fisica tra i pezzi hardware e le relazioni tra macchine fisiche e processi software, con l'indicazione dei vari punti in cui viene eseguito il codice

Gli elementi base di questi diagrammi sono:

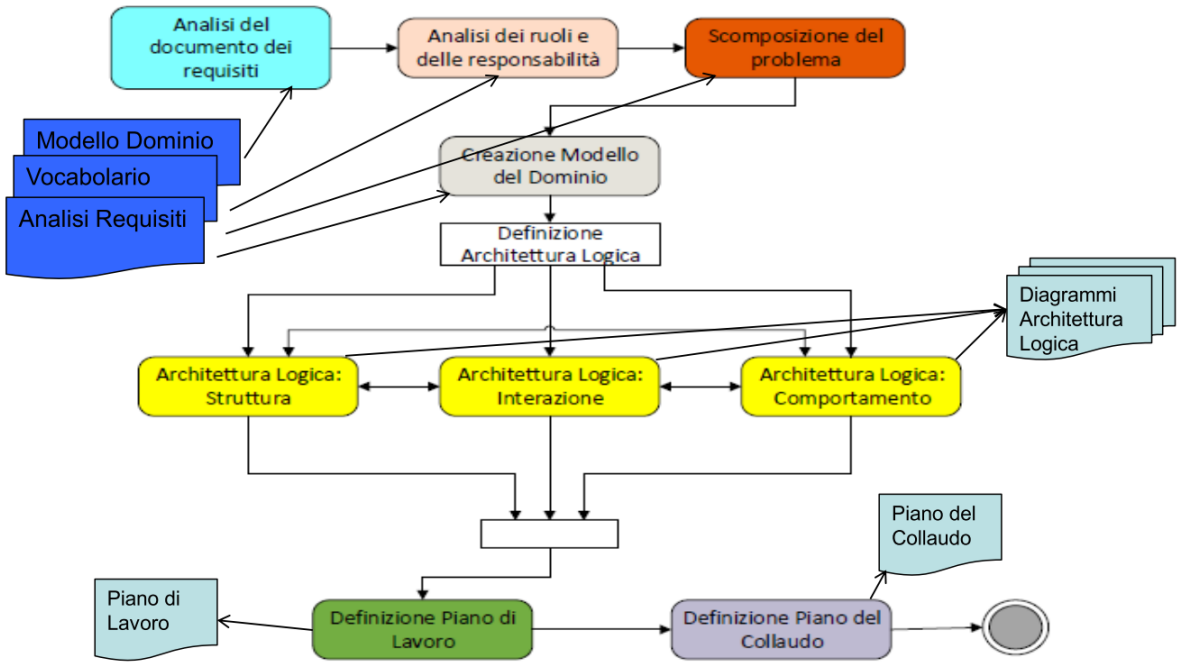
- **Nodi**: unità su cui risiedono gli artefatti (indicato tramite la relazione di tipo "deploy"). Essi comunicano tra di loro tramite dei percorsi prestabiliti.
- **Artefatti**: rappresentano una specifica porzione fisica di informazioni utilizzata o prodotta dal processo di sviluppo del software
- **Dispositivi**: è una risorsa fisica computazionale con capacità elaborative sulla quale possono essere allocati artefatti per l'esecuzione

Analisi del problema

L'obiettivo dell'analisi del problema è esprimere fatti il più possibile "oggettivi" sul problema focalizzando l'attenzione su sottosistemi, ruoli e responsabilità insiti negli scenari prospettati durante l'analisi dei requisiti senza descrivere la sua possibile soluzione.

Input: modello di dominio, vocabolario, analisi dei requisiti

Output: architettura logica, piano di lavoro, piano di collaudo



Analisi del documento dei requisiti

Partendo dai documenti dati in input occorre analizzare:

- le *funzionalità* : partendo dall'analisi del documento dei requisiti si evidenzia il tipo e la complessità delle funzionalità espresse nei casi d'uso, quali siano i dati coinvolti ed il flusso delle informazioni per ogni funzionalità

Tabella Funzionalità

Funzionalità	Tipo	Grado Complessità	Requisiti Collegati
nome funzionalità come compare nei casi d'uso	tipo della funzionalità	indicare se complessa o semplice	Requisiti a cui è collegata la funzionalità

Tabella Informazioni / Flusso

Informazione	Tipo	Livello di riservatezza/ privacy	Input/Output	Vincoli
nome (o id)	composto/ semplice	grado di riservatezza richiesto	specificare se input o output	eventuali vincoli sui valori attesi

- i *vincoli* di sistema : partendo dall'analisi dei requisiti non funzionali vanno specificati la tipologia e le funzionalità su cui impatta

Tabella Vincoli

Requisito	Categorie	Impatto	Funzionalità
Requisito non funzionale	tipo	indicare il tipo di impatto	funzionalità coinvolte

- le *interazioni* : vanno distinte le interazioni con gli umani da quelle con sistemi esterni.

Nel caso di interazioni con gli umani vanno analizzate le eventuali interfacce/maschere e creare un legame con le informazioni e le funzionalità precedentemente definite:

Tabella Maschere

Maschera	Informazioni	Funzionalità
Nome della maschera	indicare le informazioni gestite nella maschera	funzionalità coinvolte

Nel caso di sistemi esterni occorre analizzare ai morsetti i sistemi con cui si dovrà interagire ed individuare i protocolli di interazione:

Tabella Sistemi Esterni

Sistema	Descrizione	Protocollo di Interazione	Livello di Protezione
Nome del sistema	Descrizione del sistema e delle sue principali funzionalità	Specificare il protocollo di interazione richiesto	Specificare il livello di protezione garantito dal sistema

Analisi dei ruoli e delle responsabilità

Per ogni attore individuato nei casi d'uso (ruolo) si specificano le responsabilità (funzionalità), le maschere che può visualizzare, il suo livello di riservatezza, il numero di persone che possono ricoprire quel ruolo, le informazioni a cui può accedere e in che modo può accederle:

Tabella Ruoli

Ruolo	Responsabilità	Maschere	Riservatezza	Numerosità
Nome del ruolo	indicare le responsabilità assegnate	Indicare le maschere che devono essere visualizzate	Livello di riservatezza necessaria	Indicare la numerosità massima

Tabella Ruolo-Informazioni

Ruolo	Informazione	Tipo Accesso
Nome del ruolo	informazione	Specificare il tipo di accesso

Scomposizione del problema

Partendo dall'analisi delle funzionalità del sistema, permette di scomporre ogni funzionalità complessa in più funzionalità semplici, specificando quali siano i legami tra di esse

Tabella Scomposizione Funzionalità

Funzionalità	Scomposizione
nome funzionalità	elenco delle sotto-funzionalità

Tabella Sotto-Funzionalità

Sotto-Funzionalità	Sotto-Funzionalità	Legame	Informazioni
nome sotto-funzionalità	nome sotto-funzionalità	specificare il tipo di dipendenza/legame logico	specificare le informazioni scambiate

Creazione modello del dominio

Partendo dal glossario definito nell'analisi dei requisiti e dalle tabelle informazioni/flusso si individuano quali dovranno essere le classi all'interno del sistema, specificandone anche le relazioni, le associazioni, le collaborazioni («use»), gli attributi e le operazioni

Definizione architettura Logica

- **Struttura**

  La struttura viene definita tramite l'uso dei diagrammi dei package e tramite diagramma delle classi (per fornire una visione dettagliata dei package). L'architettura logica va organizzata secondo il pattern **Boundary-Control-Entity** (BCE). Esso specifica la divisione delle entità in tre categorie:

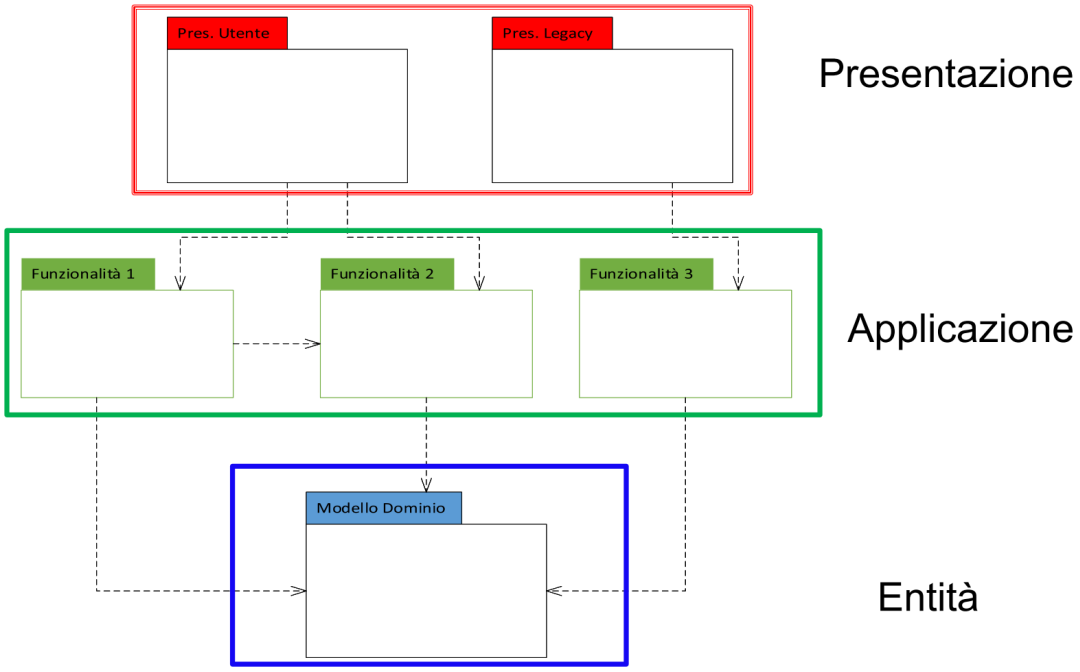
  - **Entity** : insieme delle classi che descrivono le funzionalità relative alle informazioni del sistema
  - **Boundary** : incapsulano l'interfaccia del sistema verso il mondo esterno e contengono le funzionalità per interagirvi
  - **Control** : enti che incapsulano il controllo, fanno da collante tra interfacce ed entità

  Per via di questa separazione è naturale che l'architettura risulti articolata in una sequenza di livelli (**layer**) verticali:

  - livello **presentazione** : comprende le parti che realizzano l'interfaccia utente
  - livello **applicazione** : comprende le parti che provvedono ad elaborare l'informazione di ingresso, a produrre i risultati attesi e a presentare le informazioni in uscita
  - livello **entità** : il modello del dominio applicativo

  Si parte dalla definizione dei package: prima il modello di dominio (entity), poi per le diverse funzionalità (control) e poi i package boundary con le varie dipendenze logiche passando poi ad individuare le classi che compongono ogni package facendo attenzione a non introdurre scelte di progettazione

  > *Il Package del Dominio è già stato identificato nel Modello del Dominio.*



- **Interazione**: tramite l'uso dei diagrammi di sequenza vengono specificate le interazioni tra i vari oggetti e l'ordine delle scambio di messaggi tra gli oggetti.

  > *Non è necessario spingere la definizione dei diagrammi di sequenza sino ai minimi dettagli.*
  >
  > *Utilizzare i diagrammi solo per descrivere il funzionamento del sistema.*

- **Comportamento**: descrive il comportamento interno degli oggetti tramite diagrammi di stato e delle attività

Definizione del Piano di Lavoro

definire chi fa cosa e quali sono i tempi di realizzazione e elaborazione

Definizione del Piano del Collaudo

definisce l'insieme dei risultati attesi da ogni entità definita nell'Architettura Logica in relazione a specifiche sollecitazioni stimolo-risposta prevista. Strumenti tipici per scrivere i testcase:

- JUnit nel mondo Java
- NUnit nel mondo C#

Progettazione per la sicurezza

La scelta dell'architettura influenza la sicurezza di un sistema, infatti un'architettura inappropriata non permette di garantire la riservatizza e l'integrità dei dati o il livello di disponibilità richiesto. Occorre progettare la protezione del sistema per proteggerlo dagli attacchi e la sua distribuzione per minimizzare gli effetti di un attacco riuscito. Ponendo tutti i beni in un unico posto diventa relativamente semplice proteggerli, ma in caso di un attacco andato a buon fine saranno tutti compromessi; distribuendoi beni la protezione diventa più complessa, ma un attacco andato a buon fine impatterà su meno risorse.

- **architettura a layer**: fornisce un ottimo grado di protezione. I beni critici sono posizionati nel livello più basso e sono separati dal mondo esterno da un numero di livelli dipendente dal tipo di applicazione. Ogni layer può essere protetto da delle credenziali, magari diverse tra loro
- **architettura client-server**: la protezione dei dati viene demandata alla macchina server, a cui ci si connette tramite dei client appositi. Il problema è che questa architettura è vulnerabile ad attacchi di tipo DoS, che però può essere prevenuto sfruttando la replicazione dei dati

In generale si possono seguire delle linee guida per la progettazione dell'architettura di un sistema:

1. **Basare le decisioni della sicurezza su una esplicita politica** (policy di sicurezza): questo documento deve definire i meccanismi per fornire e per far rispettare la sicurezza. Inoltre definisce quali siano le modalità attraverso cui sarà possibile accedere ai dati, quali siano le precondizione per poter accedere ai dati, e chi può accedere a quali dati. Tipicamente le policy vengono espresse come un insieme di regole in incorporate in un componente di sistema deputato a far rispettare tali politiche all'interno dell'applicazione
2. **Evitare un singolo punto si fallimento**
3. **Fallire in modo certo**: è necessario che i fallimenti, qualora accadano, siano controllati. Infatti, per evitare di fornire agli attaccanti un meccanismo di accesso privilegiato, le procedure di fall-back devono essere sicure almeno quanto il sistema stesso
4. **Bilanciare sicurezza e usabilità**
5. **Essere consapevoli dell’esistenza dell’ingegneria sociale**
6. **Usare ridondanza e diversità riduce i rischi**
7. **Validare tutti gli input**
8. **Dividere in compartimenti i beni**
9. **Progettare per il deployment**: sebbene la configurazione di un sistema siano a carico dell'amministratore di sistema, i progettisti hanno la responsabilità di progettare il deployment dell'applicativo, fornendo supporti che riducano le probabilità di errori durante la configurazione e fornendo una configurazione base sufficientemente solida.
10. **Includere supporto per visionare ed analizzare le configurazioni**: questo viene fatto ad esempio tramite programmi di utilità che consentano una configurazione più agevole del sistema e che consentano di visualizzarne lo stato corrente
11. **Minimizzare i privilegi di default**: questo limita i danni di un possibile attacco sferrato al momento dell'installazione del software. Ad esempio le credenziali di default dell'amministratore dovrebbero concedere solamente l'accesso al modulo per cambiarle e non a tutto il sistema, costringendo l'amministratore a cambiarle immediatamente
12. **Localizzare le impostazioni di configurazione**: questo fa sì che sia più semplice per chi amministra il sistema trovare le impostazioni comuni
13. **Fornire modi per rimediare a vulnerabilità di sicurezza**: questo comprende verifiche automatiche sulla disponibilità di aggiornamenti o anche il download automatico di essi. Limita la probabilità che su di un sistema rimangano vulnerabilità non patchate
14. **Progettare per il ripristino**: il sistema va progettato assumendo che potrebbero accadere degli errori, e quindi occorre predisporre delle procedure per il ripristino (associate magari a dei backup)

Security testing

Una volta sviluppato il software, oltre ai test funzionali, vanno eseguiti anche i test relativi alla sicurezza.

- **Black Box Testing**: i tester si comportano come farebbe un attaccante, cercando informazioni sulla struttura interna e procedendo per tentativi al fine di violare il sistema. Si possono impiegare anche strumenti appositi che indagano e scansionano in automatico i sistemi per prendere di mira non solo le debolezze derivanti dal codice del programma stesso (errori di programmazione, problemi legati al linguaggio usato per l'implementazione), ma anche debolezze infrastrutturali o di configurazione del sistema nel suo complesso o della rete (errori nella configurazione dei firewall, delle porte, ...)
- **White Box Testing**: viene svolto operando una revisione completa del codice e delle configurazioni ed ha come assunzione base la completa conoscenza del sistema. Sulla base delle debolezze scoperte possono essere scritti test volti a valutare le conseguenze di un attacco andato a buon fine. Tipicamente gli errori trovati riguardano la mancanza di validazione dei parametri in ingresso problemi legati all'utilizzo della memoria

Capacità di sopravvivenza

Con capacità di sopravvivenza di un sistema si intende la capacità di continuare a fornire servizi anche mentre è sotto attacco o dopo che parti del sistema sono state danneggiate a causa di un attacco andato a buon fine. Essa è una proprietà del sistema nel suo complesso e non solo delle sue parti, pertanto deve essere progettata assieme alla sicurezza del sistema.

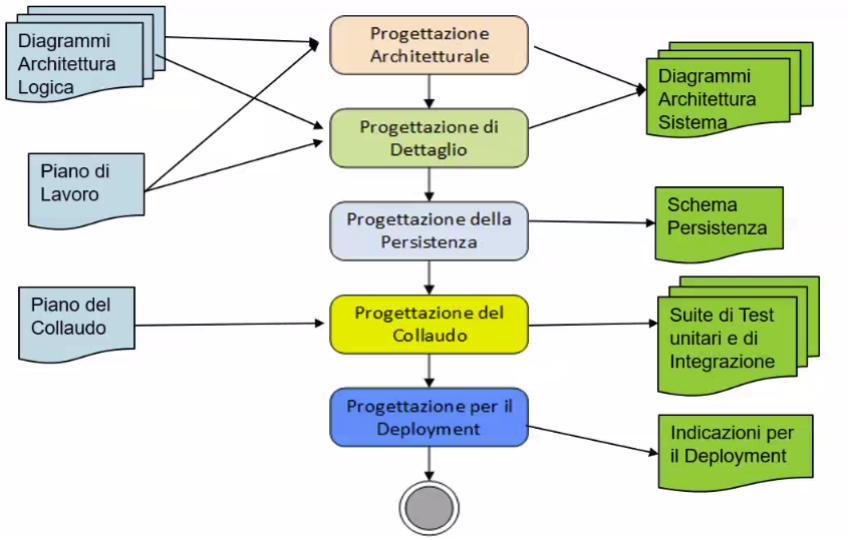
Perchè un sistema sopravviva è necessario che i servizi principali facciano altrettanto, perciò occorre identificare quali essi siano, comprendere quale sia la qualità minima del servizio erogato che deve essere mantenuta, comprendere come questi servizi potrebbero essere attaccati e quali contromisure prendere ed infine progettare il sistema per un ripristino rapido

L' analisi della sopravvivenza comprende quattro fasi cicliche:

1. **Comprensione del sistema**
2. **Identificazione dei servizi critici**
3. **Simulazione degli attacchi**
4. **Analsi della sopravvivenza**: permette di identificare quali siano i componenti messi a rischio dall'attacco e quali strategie si possono adottare per resistervi

Progettazione

- **Progettazione**: attraversi raffinamenti successivi dell'architettura logica permette di raggiungere un'architettura di sistema definitiva. Non solo ha come scopo quello di fornire una soluzione ai problemi, ma soprattuto spiegare le motivazioni che hanno spinto ad adottare tale soluzione



- **Progettazione Architetturale**: dopo un'attenta valutazione dei requisiti non funzionali (quali prestazioni, protezione, disponibilità, manutenibilità, ...) e dei requisiti relativi alla sicurezza occorre scegliere quale sia l'architettura più adatta per il sistema. Esempi notevoli sono:

    - **Layer**: iuta a strutturare quelle applicazioni che possono essere scomposte in gruppi di sotto-attività in cui ciascun gruppo si trova ad un ben definito livello di astrazione
    - **MVC**
    - **Client/Server**
    - **Broker**: può essere usato per strutturare sistemi distribuiti con un disaccoppiamento tra i diversi sotto-sistemi che comunicano tra loro attraverso remote server invocation; il broker fa da coordinatore per le comunicazioni
    - **Pipes & filters**: aiuta a strutturare quelle applicazioni che processano stream di dati (usato ad esempio negli stream Java)

  Occorre inoltre scegliere l'architettura del sistema. L'architettura scelta influenza:

    - prestazioni - l'architettura dovrebbe essere progettata localizzando le operazioni critiche all'interno di un piccolo numero di componenti grandi minimizzando le comunicazioni possibili tra essi.
    - protezioni dei dati (security) - l'architettura dovrebbe essere progettata a strati, collocando le risorse più critiche nello strato più interno
    - sicurezza (safety) - l'architettura deve essere progettata con un piccolo numero di componenti grandi
    - disponibilità - l'architettura deve essere progettata con tanti componenti piccoli così che se uno smette di funzionare il sistema complessivamente può ancora funzionare parzialmente
    - manutenibilità - l'architettura deve essere progettata usando componenti piccoli, atomici, con poche dipendenze
- **Progettazione di Dettaglio**: definisce nel dettaglio l'architettura del sistema, sia per quanto riguarda le entità in gioco (uso dei design pattern, specifica della navigabilità tra le associazioni, principi di progettazione, ...)che per quanto riguarda le interazioni tra di esse. La progettazione dovrebbe essere il più slegata possibile dalle scelte tecnologiche.
- **Progettazione della Persistenza**
- **Progettazione del Collaudo**
- **Progettazione per il Deployment**: permette di specificare quale sia l'architettura su cui girerà il prodotto software. Questo viene specificato tramite i *diagrammi del deployment*